Worst case analysis of TreeMap data structure

Frédéric Fauberteau, Serge Midonnet

Worst Case Analysis of TreeMap Data Structure

Frédéric Fauberteau and Serge Midonnet
Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge,
UMR CNRS 8049, France
{fauberte, midonnet}@univ-paris-est.fr

Abstract

Data structures with relaxed balance eases the update of shared resources on asynchronous parallel architectures. This improvement is obtained by a better locking scheme of the data structure. In this paper, we describe the complexity and analyze the worst case cost of access operations on such a structure. We propose a data structure with the same properties as Java TreeMap but implemented with chromatic search tree; a tree with relaxed balance. The aim of our structure is to provide a more efficient TreeMap we can use in concurrent and real-time applications.

1. Introduction

A chromatic search tree, initially presented in [6], is a red-black tree [1] with relaxed balance. A red-black tree is a self-balancing binary search tree. This kind of structure allows to perform update operations (insertion and removing) and search operations with a $log_2(n)$ time complexity in the worst case where n is the number of keys in the tree. When such a tree looses its balance after an update, a balancing operation is performed to guarantee the height of the tree and so the $log_2(n)$ complexity. A chromatic search tree is a relaxed balance tree. Rebalancing in a chromatic search tree has been studied in [9, 3, 10]. In [2] Boyar, Fagerberg and Larsen prove that only an amortized constant amount of rebalancing is necessary after an update in a chromatic search tree depending on the number of successive operations. In [7] Hanke compares red-black trees performances with their relaxed versions and gives simulation results which show that chromatic search trees offer better performances than red-black tree in concurrent environment.

With the emergence of the multiprocessor architectures in the real-time issues, data structures which are more efficient in concurrent context could be of value. To decide the faisability of a real-time application, a worst case execution time estimation of all tasks which compose the system must be carried out. This estimation implies, if tasks are using a data structure concurrently, knowing the worst case cost of the use of such a structure. As far as we know, nobody else has, as yet, looked at the estimation cost of the update or search operations on a chromatic search tree.

The remainder of this paper is organized as following. In Section 2, we present red-black and chromatic trees and we describe their properties. In Section 3, we propose a worst case estimation of operation cost for the two structures. In Section 4, we explain how we want to use idle time of the system to rebalance the chromatic search tree. In Section 5, we comment on the results of our simulations. Finally, in Section 6, we conclude and discuss our future research work.

2. Red-black and Chromatic Search Trees

Red-black trees and their relaxed equivalent are binary search trees. These structures are used for store datas in a sorted way. Datas are stored in the nodes of the tree. They must be comparable to be stored in such a structure (otherwise a comparable key can be associated to the data). Each node v in a red-black tree or in a chromatic search tree has an associated non-negative weight w. This weight is used for guarantee the balance of the tree. If $w(v) = 0$, the node is red; if $w(v) = 1$, the node is black. In a chromatic search tree, a node can be overweighted and its weight can be $w(v) > 1$. The weighted level of a node is the weight of the path from the root to that node.

Definition 1. A red-black tree is a full binary search tree T with the following balance conditions:

- the leaves of T are black,
- all leaves of T have the same weighted level,
- no path from a root to a leaf contains two consecutive red nodes,
- T has only red and black nodes.

A chromatic search tree is a leaf-oriented tree. This property enables the user to insert a new key locking the leaf where this key must be inserted and allows to remove a key locking the parent of leaf which contains this key.

Definition 2. A chromatic tree is a full binary search tree T with the following conditions:

- the leaves of T are not red,
- all leaves of T have the same weighted level.

3. Worst Case Estimation

In this section, we compare red-black tree and chromatic search tree. We firstly estimate the worst case execution time for the two structures and we propose a bound on rebalancing the chromatic search tree. Secondly, we compare the blocking factor of tasks using these structures and show the advantage of the chromatic search tree. We consider only the *insert* operations because *remove* and *search* operations have an equivalent worst case execution time for two structures.

3.1. Without Shared Resource

Firstly, we consider the worst case execution time of a task without shared resource. This task uses a binary search tree of maximum size N. The worst case for this task is the insertion of the N^{th} key in the tree. For the red-black tree, Wood [12] give upper bound for the height of the tree:

$$h_{RB} \leq 2\log_2(N+2) - 2 \tag{1}$$

An insertion in a red-black tree can consist in the search of the position where to insert the new key followed by a fix operation. The fix operation can consist in color modifications and two rotations in the worst case. The tree is browsed from the new leaf to the root. The color of the nodes on this path is updated function of the color of their parent nodes. In the same way, if a node on this path matches some conditions on its parent nodes, a rotation is performed. All the fix operations are precisely described in [4]. The two rotations have a constant cost but the modifications of the color of the nodes along the tree are dependent of the tree height. We define the following costs:

- C_k the cost of a key comparaison,
- C_a the cost of a node insertion (connection of pointers),
- C_c the cost of a color modification,
- C_r the cost of a rotation.

We can give a pessimistic worst case estimation time for an insertion in a red-black tree of size N with the following equation:

$$(C_k + C_c) \cdot 2\log_2(N+1) - 2 + C_a + 2C_r \tag{2}$$

For a chromatic search tree, the worst case is when the tree has never been rebalanced. So, the insertion of N^{th} key in tree is equivalent to the insertion of the key at the end of a list. This insertion is followed by a unique color modification. We can also give a worst case estimation time for an insertion in a chromatic search tree of size N with the following equation:

$$C_k \cdot (N-1) + C_a + C_c \tag{3}$$

3.2. With Shared Resource

In this section, we consider a set of tasks which share a resource represented by our binary search tree. We also consider that the Priority Inheritance Protocol is used because it is commonly implemented on the actual systems. With such a protocol, a task of high priority τ_H can be blocked by a task of lesser priority τ_L while τ_L is locking the resource. Because a balance operation in a red-black tree can modify the internal structure of the tree, it is imperative to fully lock the structure. We note here that C_i the WCET given by Equation 2. If τ_L performs I insertions with a red-black shared resource, the blocking factor of τ_H is given by the following equation:

$$I \cdot C_i \tag{4}$$

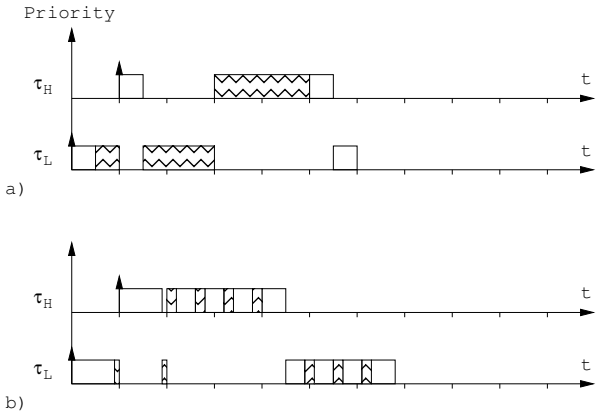

Figure 1. a) Red-black tree shared resource. b) Chromatic search tree shared resource.

We represent in Figure 1 two tasks which share a resource. The dashed parts represent the critical sections. In Figure 1-a the task τ_H which is blocked by τ_L during τ_L perform all its insertions.

Now we consider that the shared resource is a chromatic search tree. This structure is not balanced at each update operation. So the internal structure is never modified. A task which uses this resource just needs to lock the leaf where the new key must be inserted. If τ_L performs I insertions with a chromatic search tree shared resource, the blocking factor of τ_H will be in the worst case:

$$C_a + C_c \tag{5}$$

This worst case occurs when τ_H tries to insert a key in the same leaf that τ_L has blocked. In this case, τ_H just waits for τ_L insert its key, modify the color of the old leaf and unlock it. We represent this scenario in the Figure 1-b.

The chromatic search tree was studied to reduce locks in shared-memory and we show that it enables the reduction in response time of the high priority tasks when priority inheritances occur.

4. Rebalancing using Slack Stealer

Rebalancing time depends of the number of update operations and of the size of the tree. We know these parameters, so we can estimate the necessary time needed to rebalance a chromatic search tree. Because the rebalancing operation has been removed from the task cost, we propose to perform it during the idle time of the system. Several methods are possible to perform a treatment during idle time:

- scheduling of an aperiodic task of high priority when rebalancing operation is needed. The drawback of this method is that the rebalancing operation can consume some time which prevents some periodic tasks to meet their deadline,
- scheduling of a task server. This method implies the reservation of server cost, but rebalancing may not always be needed. Thus this method adds a further task to the feasibility analysis of the system (the task server) and complexifies this analysis,
- computing the available slack when the rebalancing operation is needed.

We have chosen the last method because the slack gives us the maximum duration for which all tasks can be delayed without missing their deadline. So we can perform the rebalancing operation during this time and interrupt it when all the slack has been consumed because each rotation operation of the rebalancing processing is atomic.

The computation of the slack can be perform with a *Slack Stealer* algorithm. There are two kind of optimal algorithms which give the exact slack value for a t instant. First, [11] propose a static approach to compute the slack. When the system is offline, it constructs a function representing the value of the slack for an hyperperiod. When the system is online, this function is updated to keep the value of the slack after its consumption by the tasks. The drawback of this algorithm is that the space complexity of the storage of the function is too large. A second approach, proposed in [5], compute the slack online without precomputation. The disadvantage of this algorithm is that time complexity of the computation is too large. These optimal algorithms are not exploitable to implement in a real-time application. Finally, a particular kind of algorithm computes an approximation of the slack value in $O(1)$. We can use the algorithm proposed in [8] to compute the slack value when the rebalancing is needed. This algorithm adds a linear time complexity operation to the WCET of each task.

5. Results of Simulation

We present in this section our first results. We produce these results implementing a new map structure based on the Java TreeMap. We replaced the red-black tree by a chromatic search tree to produce a structure we called RTChromaticTreeMap.

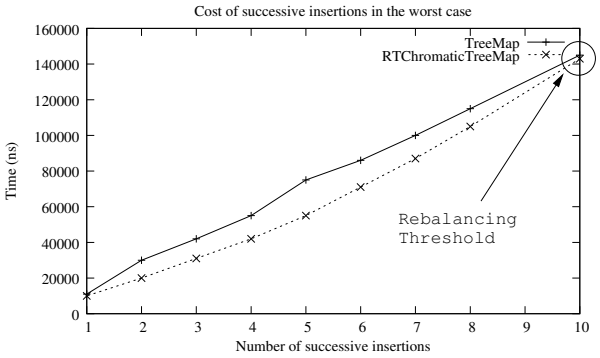

Figure 2. Worst successive insertions in TreeMap and RTChromaticTreeMap.

We represent in Figure 2 measures of successive insertions in the both structures. We have initialized the two structures with a little set of keys. This particular set of keys produces a balanced tree for both structures. From these initialized trees, we perform insertions of sorted values which must be inserted in the same leaf. For the RTChromaticTreeMap, we don't apply the rebalancing algorithm. We remark that RTChromaticTreeMap has better performances for a little number of successive insertions. We obtain this results because in the worst case situation in which the inserted key is always in the same leaf, the TreeMap structure operates rebalancing operation more frequently. In the RTChromaticTreeMap, a list is created from the leaf where the new key

is inserted. Even if browsing this list represents an overcost, it is counterbalanced by the lack of rebalancing operation. But after 10 successives insertions, the overcost of the RTChromaticTreeMap become more important than the cost of rebalancing operation of the TreeMap. So, we have used this value as a first rebalancing threshold.

In the worst case execution time evaluation, we must consider the cost of the worst rebalancing operation for each insertion. But in an average case, a rotation operation can not be necessary. In this case the overcost of the RTChromaticTreeMap is not counterbalanced.

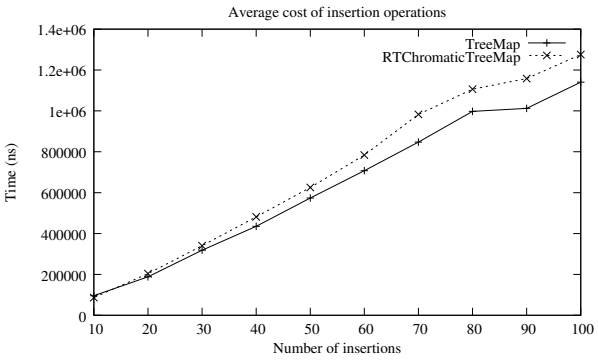

Figure 3. Random successive insertions in TreeMap and RTChromaticTreeMap.

Tor compare the performances of the two structures in a less pessimistic context, we measured of average insertions. We can show in Figure 3 that the cost of insertion operations in RTChromaticTreeMap is slightly greater that the cost of insertion operations in TreeMap. So, we can improve the worst case execution time estimation but the average performance of the RTChromaticTreeMap structure are not better. This structure must be used only if there is concurrence between tasks in the system.

6. Conclusion and future Work

We have shown that chromatic search tree can be used in a real-time context. But to reduce the worst case execution time, it is necessary to fix a bound to force the rebalancing of the structure. We assume that we can perform the rebalancing during idle time using *Slack Stealer* algorithm. That supposes that the system is underloaded.

We have implemented a Java TreeMap structure with chromatic search tree. We must now implemente the local locking scheme and the approximate *Slack Stealer* algorithm to finalize our tests and propose an exploitable solution.

References

[1] R. Bayer and E. M. Mccreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[2] J. Boyar, R. Fagerberg, and K. S. Larsen. Amortization results for chromatic search trees, with an application to priority queues. *J. Comput. Syst. Sci.*, 55(3):504–521, 1997.

[3] J. Boyar and K. S. Larsen. Efficient rebalancing of chromatic search trees. *J. Comput. Syst. Sci.*, 49(3):667–682, 1994.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, September 2001.

[5] R. I. Davis. Scheduling slack time in fixed priority preemptive systems. Technical report, Dec. 08 1993.

[6] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *FOCS*, pages 8–21. IEEE, 1978.

[7] S. Hanke. The performance of concurrent red-black tree algorithms. In J. S. Vitter and C. D. Zaroliagis, editors, *Algorithm Engineering*, volume 1668 of *Lecture Notes in Computer Science*, pages 287–301. Springer, 1999.

[8] D. Masson and S. Midonnet. Slack time evaluation with rtsj. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 322–323, New York, NY, USA, 2008. ACM.

[9] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *PODS*, pages 192–198. ACM Press, 1991.

[10] O. Nurmi and E. Soisalon-Soininen. Chromatic binary search trees. a structure for concurrent rebalancing. *Acta Inf.*, 33(6):547–557, 1996.

[11] S. R. Thuel and J. P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. In *IEEE Real-Time Systems Symposium*, pages 22–33. IEEE Computer Society, 1994.

[12] D. Wood. *Data structures, algorithms, and performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.